

Chapter 5: Creating user functions

User functions, an advanced feature of Meridian IVR, allow you to interface “C” language code to a Meridian IVR application to perform activities which the ordinary cell types cannot do.

Up to 10 input buffers, along with one function code, can be passed into user functions. There are 99 different function codes; therefore, each user function can perform up to 99 different functions. Up to 10 output buffers, along with 10 different reply codes, can be returned from each user function. These reply codes allow different branches to be taken depending on the response received from the user function.

To utilize user functions effectively, you should be familiar with “C” programming language and the operating system running on the Meridian IVR application module. You should understand the procedures presented earlier in this manual for designing and building applications.

This chapter covers the following topics:

- “Understanding the user function process” on page 5-2
- “User-defined functions” on page 5-5
- “Understanding the importance of time-out” on page 5-7
- “Testing your user function” on page 5-10
- “Including the user function in an application” on page 5-12
- “Using the usr.c file as a template” on page 5-13



CAUTION! **Risk of data loss**

Do not develop “C” language code on a live system as it is very powerful, and can be potentially dangerous if poorly written.

Understanding the user function process

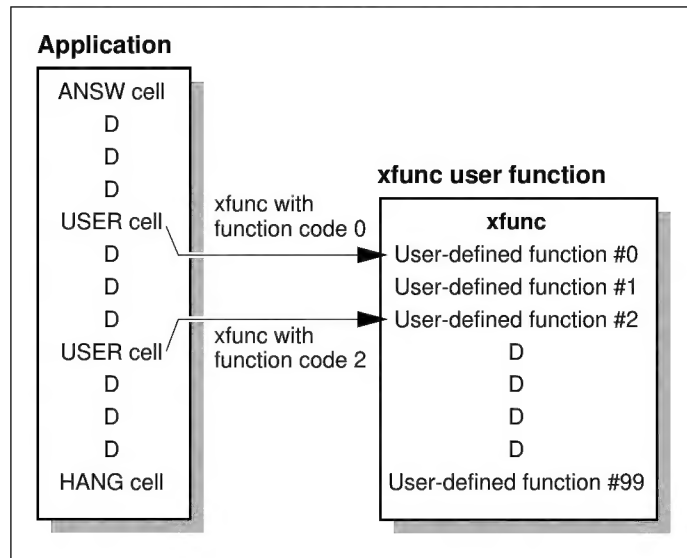
Before you code your own user functions, you should understand how an application interacts with a user function and how the process that is running (herein called “the user function process”) is shared with the other channels on the system.

Accessing the user function from the call flow

When you code a user function, you also define specific activities which the user function can perform. Each activity, or user-defined function, is identified by a two-digit function code. Your user function is referenced in the call flow of an application by using the USER cell. The USER cell references the user function, and identifies the activity the user function performs.

Figure 5-1 illustrates an application that passes control to the user function “xfunc” through the USER cell. In the first instance, xfunc executes user-defined function #0. In the next instance, xfunc executes user-defined function #2.

Figure 5-1
Application passing control to user function



The user function process

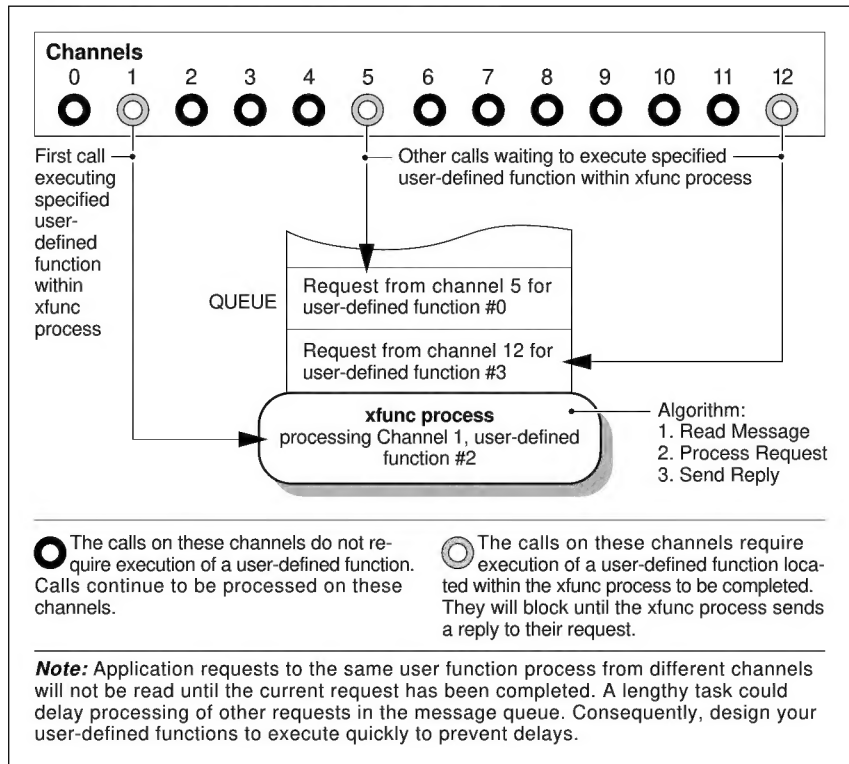
Only one user function process operates at a time regardless of how many running applications require it. If an application references a user-defined function located within the xfunc user function, the xfunc process starts automatically when you load the application.

If an application must execute a user-defined function from a user function process that is already running, that application uses the existing user function process rather than starting up a new one. This process is shared with all the available channels and does not perform any tasks until it receives a request in the message queue.

When you load and execute an application that references a user function, the following sequence of events occur:

- 1 Your application starts.
- 2 When the application executes a USER cell, it passes the corresponding user function process a message that includes a set of *input buffers* and a *function code* to indicate which user-defined functions should be executed.
- 3 The USER cell then transfers control of the call flow to the user function itself, and the application on the associated channel is suspended.
- 4 The user function process begins executing the specified user-defined function that is identified by a function code.
- 5 Meridian IVR continues processing calls on the other channels (see [Figure 5-2](#)). When the user function process has processed your task, it passes the application a set of *output buffers* and a *reply code*, then returns control to the application.

Figure 5-2
How the user function process works



- 6 The application branches to the next cell in your call flow application. The next cell is determined by the value of the reply code.
- 7 When you unload all applications that reference the same user function, the user function process is killed.

User-defined functions

Specific user functions can be referenced by applications running on many channels; however, only one channel can access the corresponding user function process at any given time. This means that the user function must finish whatever it is doing quickly each time it is called to avoid delays.

Input and output buffers

Input and output buffers provide a flexible way to transmit data between the application and the user function. An application can send the user function up to ten input buffers which can be any of the Meridian IVR system buffers or user-defined buffers. Similarly, the user function can return up to 10 output buffers to the application. As with all buffers in Meridian IVR, each of the input or output buffers can store a null-terminated string of up to 31 ASCII characters.

Reply code

When a user function finishes its current task, it returns a reply code which can be a value from zero to nine. When the application resumes, the USER cell checks the reply code to determine which cell the application should execute next. The USER cell has up to ten branches, and it uses the reply codes, found in the `usr.c` template file, to determine which of these branches to take.

If you want to force an error code, you need to define the codes in the user function by setting the reply code variable before returning to your application. Your application can check this reply code value and take a branch based upon this value. This way, the user cell passes the reply code set into the call flow and allows your application to choose which next cell to execute.

Source file for creating user functions

A source file, called “`usr.c`”, is included in the Meridian IVR software. You can use this file as a template to create user functions. This template contains the code you need to initialize and operate a user function process. Copy this file and insert your own code at a few specified points to create a user function. For a complete listing of the `usr.c` file, read the section [“Using the `usr.c` file as a template” on page 5-13](#).

While your application is executing, your user function process reads the message sent from the USER cell, determines which user-defined function has been selected, executes the user-defined function, updates output buffers as necessary, and sends a message back to the USER cell with a reply code.

Error handling

Whenever a user-defined function encounters an error, it writes an error message to the event log. A user-defined function can send error and informational messages to the transaction log through calls to “post_log_msg()”, which takes a message string as its only argument. You can run the Transaction Log Report to view the error messages.

Understanding the importance of time-out

Calls can sometimes become stuck within a user function. This can prevent the call from progressing and result in hung calls and frustrated callers. However, you can code all Meridian IVR user functions with a time-out parameter to eliminate this inconvenience.

The time-out input parameter allows the application to take a time-out branch. In Meridian IVR Release 1.2, the treatment of time-outs is left completely up to custom code which the application developers must write.

For Meridian IVR Release 2.0/I, the user function template for the customized user functions is enhanced to include a time-out parameter. If you specify an input time-out value, the application is delayed in the user function for the time-out value you specify. The application then takes the time-out branch.

For any complex user functions, especially those which interface to external hosts, we strongly recommend that the application developers alter time-out treatment to meet their specific needs.

ATTENTION!

Be sure that you develop user functions properly.

User functions that crash or hang a channel cause the application to not take the time-out branch (or any other branch) from the user function cell.

Procedure 5-1
Designing a user function

- 1** Decide what you want the user function to do. Break its activities into separate functions that can be written as individual user-defined functions.

- 2** Choose a name for the user function.

The name can be up to five characters long. Do not use any of the following names since they conflict with existing components in Meridian IVR:

cli	vtk	larx
dbx	vid	csnap
dcm	veh	sae
ust	psm	scm
vrtd	vbm	smi
csc	trs	snap
sam	access	bpe
qds	msg	xae
sde	mm	sri
lh	vrn	uel
sad	xai	sai
log	tmr	vtf

- 3** Decide what buffer information should be passed to each user-defined function.

- 4** Decide what buffer information should be returned to the application.

A maximum of ten input buffers and ten output buffers can be used each time a USER cell invokes the user function.

- 5** Decide which reply code each user-defined function returns under various circumstances.

The user function returns a reply code which the USER cell utilizes to go to one of ten possible branches.

- 6 Decide how each user-defined function can determine that an error has occurred, and how it can return an error reply code.
- 7 Plan error messages and informational messages to cover all the possibilities.
- 8 Decide how each user-defined function will make calls to `post_log_msg()` to pass error strings.

ATTENTION!

We highly recommend that your user functions do not call `printf()`. Instead, use the error log to avoid interfering with the normal operation of the Meridian IVR system administration.

Once you have completed these steps, you are ready to code a user function.

As you read these instructions, refer to the `usr.c` source code in the section **“Using the `usr.c` file as a template” on page 5-13.**

Procedure 5-2 **Coding a user function**

- 1 Change the working directory to `/user/ivr/gen/usr`.
- 2 Copy and rename the `usr.c` file.

The new name identifies the name of your user function. It can be up to five characters long. For example, to create a user function called “`xfunc`”, copy `usr.c` to `xfunc.c`. Copying this file, instead of editing it directly, ensures that you have an unaltered source file to use when you want to create another user function.

Edit the switch statement in your *filename.c* file as follows:

- 3 Specify a “case” statement for each user-defined function that the user function can execute.
- 4 For each case statement, assign a value to the `reply_code` variable, matching one of the codes in the Reply Code table for your application. This value can also correspond to an error condition; refer to the default case statement in the `usr.c` template file.

- 5 If you want to return information to the application from the user-defined function, assign a value (1–10) to the `*p_number_of_output_buffers` variable corresponding to the number of output buffers to be passed back to the application.
- 6 Assign values to the output buffers in the `output_buffer_array` structure. These values are passed back to the application during execution.
- 7 Save your edits.

Procedure 5-3 Building a user function

- 1 In the `/u/ivr/usr` directory, type the following command:

make -e -f user.make NAME=usr_func <Enter>

For example, to build an executable file for `xfunc.c`, enter

make -e -f user.make NAME=xfunc <Enter>

Note: Since you have entered the name of your file without the `.c` extension, the resultant executable user function is `xfunc`, created in the current directory.

Testing your user function

Before you actually use your user function in an application, you should test the user function to ensure that it executes properly (for example, each case statement matches a specific user-defined function, or the appropriate number of buffers are passed). You can do this by using the user function stand-alone mode.

Procedure 5-4 Testing user function with stand-alone mode

- 1 Enter the following command at the shell prompt, where `user_func_name` is the name of the user function that you just created:
user_func_name -i <Enter>
- 2 Respond to the following prompts:

Function code The number (0–99) identifying the user-defined function and the corresponding switch-case statement in the user function you want to test. To exit the user function, type **-1**.

Current channel Enter any number from 0–63.

Number of input buffers The number of data buffers (0–10) to be passed to the user function.

Input buffers Values to be entered into each of the input buffers being used.

The user function executes, using your responses to these prompts, and displays the following information:

User function reply A code returned to the calling application. It identifies the status of the execution.

Number of output buffers The number of data buffers (0–10) that are returned to the calling application.

Output buffers The contents of the output buffers.

The following example illustrates the execution of the `usr.c` template in stand-alone mode. Type the following prompt at the shell prompt to start the “usr” user function:

usr -i

```
Enter Function Code (-1 to exit): 1
Enter Current Channel: 0
Enter Number Of Input Buffers: 0
==== about to call user function
==== returned from user function
User Function Reply: 1
Number Of Output Buffers: 0

Enter Function Code (-1 to exit): 2
Enter Current Channel: 0
Enter Number Of Input Buffers: 5
      Input Buffer #1: 1
      Input Buffer #2: 2
      Input Buffer #3: 3
      Input Buffer #4: 4
      Input Buffer #5: 5
==== about to call user function
==== returned from user function
User Function Reply: 2
Number Of Output Buffers: 5
      Output Buffer #1: '2'
      Output Buffer #2: '4'
      Output Buffer #3: '6'
      Output Buffer #4: '8'
      Output Buffer #5: '10'
```

```
Enter Function Code (-1 to exit): 3
Enter Current Channel: 0
Enter Number Of Input Buffers: 2
    Input Buffer #1: 1
    Input Buffer #2: 2
==== about to call user function
==== returned from user function
User Function Reply: 3
Number Of Output Buffers: 1
    Output Buffer #1:'3'

Enter Function Code (-1 to exit): 4
Enter Current Channel: 0
Enter Number Of Input Buffers: 0
==== about to call user function
==== returned from user function
User Function Reply: 4
Number Of Output Buffers: 0

Enter Function Code (-1 to exit): 5
Enter Current Channel: 0
Enter Number Of Input Buffers: 0
==== about to call user function
>>>> call to post_log_msg('Unknown function code: 5')
==== returned from user function
User Function Reply: 0
Number Of Output Buffers: 0

Enter Function Code (-1 to exit): -1
```

To promote user function to the “exe directory”, enter this command:

```
make -e -f user.make NAME = user_func_name promote <Enter>
```

Meridian IVR cannot execute your user function unless you have previously entered this command.

Including the user function in an application

Now that you have built a user function, you can design and build an application to call the user function. To do this, you should read the previous chapters and become familiar with the procedures for designing and building applications. For more information on each user cell in your application, read the section on user cells in Chapter 7, “Cell catalog”.

Procedure 5-5
Including a user function in your application

- 1 Enter the name of the user function in the User Function Name parameter.
- 2 Enter the function code in the Function Code parameter.
- 3 Enter the number of reply codes in the # of Reply Codes parameter.
- 4 Enter the number of input buffers in the # of Input Buffers parameter.
- 5 Enter the number of output buffers in the # of Output Buffers parameter.
- 6 List the possible reply codes and their corresponding next cells in the Reply Code table.
- 7 Go to the Input Buffer table and list the input buffers.
- 8 Go to the Output Buffer table and list the output buffers.

Once you have built the application, you can load and run it. See the *Meridian IVR System Administration Guide* (NTP 555-9001-300) for procedures, or consult your system administrator. When the application is running, you should run the Transaction Log Report occasionally to see if the user function has written any errors or informational messages. The *Meridian IVR System Administration Guide* also provides procedures for running this report.

Using the usr.c file as a template

Here is a listing of the Meridian IVR usr.c source file, which you can use as a template for creating your User Functions. The usr.c file already contains sample User Function code in the switch statement (i.e., cases 1, 2, 3, and 4, highlighted in bold). Simply remove this code and enter your own case statements. *Before you modify the code, be sure you copy usr.c to another file and work on the copy.*

```
static char usr_c[]="@(#) $Id: usr.c,vpf_main 1.4 1994/10/11
17:25:09 kwu Exp $";
/*****
*****
*   usr.c
*
*   This file contains the routine user_function().
*
*
*   WARNING:
*
*   Any include files added to this file will NOT be
*   reflected in the stated dependencies of usr.makefile.
*****/
#include <stdio.h>

#include "vtk_std.h"

#include "vpf.h"
#include "usr_prototypes.h"

#define MAX_BUFS_PASSED

/*****
*****

*   extern global variables
*/
extern FILE      *fp_verbose;           /* file pointer for
verbose mode    */
extern int       timer_expired;         /* time out ? set if
TRUE*/

/*****
*****
*   For multi-threaded user functions, the number of child
processes
*   should be specified here.
*
*   Note that zero children is the default, and means that the
user process
*   will operate in a single threaded mode as it has always
done in the past.
```

```

*****
*****
*/
int    number_of_children = 0;    /* number of children to
spawn, if any    */

/*****
*****
*-h-   int    init_user_function(max_chans)
*       int    max_chans;        - maximum possible channels
*
*   DESCRIPTION
*       User defined user_function initialization
*
*       This initialization routine is for the user to perform
certain
*       tasks before the process begins taking requests from
applications.
*
*       Note that in a multi-threaded user function. This
routine is executed
*       in each child process, but not in the parent (manager)
process
*
*   RETURNS:    TRUE    if successful

*               FALSE   otherwise

*****
*****
*/
int    init_user_function(max_chans)
int    max_chans;
{
    verbose_msg ("init_user_function");

return(TRUE);

}
/*****
*****
*-h-   void    end_user_function(max_chans)
*       int    max_chans;        - maximum possible channels
*

```

```
* DESCRIPTION
*   User defined user_function cleanup
*
*   This termination routine is for the user to perform
certain
*   tasks after the process has been notified to terminate.
*
*   Note that in a multi-threaded user function. This
routine is executed
*   in each child process, but not in the parent (manager)
process
*

*****
*****
*/
void    end_user_function(max_chans)
int     max_chans;
{
    verbose_msg ("end_user_function");

    return;

}
/*-----*/
/*-h-  int      user_function (function_code, current_channel,
**                                     number_of_input_buffers,
input_buffer_array,
**                                     p_number_of_output_buffers,
output_buffer_array)
**      int      function_code;
**      int      current_channel;
**      int      number_of_input_buffers;
**      char     input_buffer_array[MAX_BUFS][MAX_BUF_SIZE+1];
**      int      *p_number_of_output_buffers;
**      char     output_buffer_array[MAX_BUFS][MAX_BUF_SIZE+1];
**      u_long   tout;
```



```

**
**  DESCRIPTION:
**      Perform the actions of each user function.
**      Each CASE in the SWITCH statement acts as a different
user function
**      within this process
**
**  RETURNS:
**      The reply code (0 - 9) to the user function cell.
**      The cell will then branch based on this value.
*****
*****
*/
int      user_function ( function_code, current_channel,
                        number_of_input_buffers,
input_buffer_array,
                        p_number_of_output_buffers,
output_buffer_array,

                        tout)

int      function_code;
int      current_channel;
int      number_of_input_buffers;
char     input_buffer_array[MAX_BUFS][MAX_BUF_SIZE+1];
int      *p_number_of_output_buffers;
char     output_buffer_array[MAX_BUFS][MAX_BUF_SIZE+1];
u_long   tout;
{
    /* —— Variables for use by ALL user functions. —— */
    /*
        int      reply_code = 0;
        char     error_message[80];
        /* —— Variables used by THIS user function. —— */
        int      count;
        long     sum;

        /*
         * Assign a default value for the number of output buffers.
         * (Because p_number_of_output_buffers is a pointer to an
integer the
         * indirection operator ('*') must be used to assign an
integer value.)
        */
        *p_number_of_output_buffers = number_of_input_buffers;

```

```
switch (function_code)
{
    case 1:
        /* Do nothing */
        *p_number_of_output_buffers = 0;
        reply_code = 1;
        break;

    case 2:
        /* Numerically double buffer values */
        for (count=0; count<number_of_input_buffers;
count++)
        {
            sprintf(output_buffer_array[count], "%ld",
                2*atol(input_buffer_array[count]));
        }
        *p_number_of_output_buffers =
number_of_input_buffers;
        reply_code = 2;
        break;

    case 3:
        /* Numerically sum buffer values */
        sum = 0L;
        for (count=0; count<number_of_input_buffers;
count++)
        {
            sum = sum + atol(input_buffer_array[count]);
        }
        sprintf(output_buffer_array[0], "%ld", sum);
        *p_number_of_output_buffers = 1;
        reply_code = 3;
        break;

    case 4:
```

```

/* Any instruction (except those listed below) will be
interrupted when
    a timeout occurs. The timer_expired variable will be set
to TRUE. The
    User function programmer needs to check timer_expired after
every
    instruction that can block the process as the example below
shows. */

/* Note: The UNIX system() call or any child of this user
function will not
    be able to receive the timeout signal*/

start_timer(tout);      /* tout is timeout duration */
sleep(180);             /* sleep for 3 minutes */

/* A suspected timeout may occur. Check global variable
timer_expired and call timeout_function if timeout occurs */
if (timer_expired)
{
    timeout_function(function_code,
current_channel,
                                number_of_input_buffers,
                                input_buffer_array,
                                p_number_of_output_buffers,
                                output_buffer_array);

    p_number_of_output_buffers,
    output_buffer_array); return(-1);
}
    *p_number_of_output_buffers = 0;
    stop_timer();
    reply_code = 4;
    break;
default:
    /* —— Print Error if in verbose mode —— */
    if (fp_verbose)
        fprintf(fp_verbose, "\tfunction_code %d not
implemented\n",
                                function_code);

```

```
        /* ----- Report Error to error logger ----- */
        sprintf(error_message, "Unknown function code: %d",
function_code);
        post_log_msg(error_message);
        *p_number_of_output_buffers = 0;
        break;
    }

    return (reply_code);
}

/*****
*****
*-h-   int      init_user_manager(max_chans)
*      int      max_chans;          - maximum possible channels
*
*  DESCRIPTION
*      User defined user_manager initialization
*
*      This initialization routine is for multi-threaded user
functions
*      where the user needs to perform certain tasks within
the manager
*      process before it begins taking requests from
applications or
*      communicating with its children user functions.
*
*      Note that this routine only executes if
number_of_children > 0
*
*  RETURNS:   TRUE    if successful
*             FALSE   otherwise
*****
*****
*/
int      init_user_manager(max_chans)
int      max_chans;
{
    verbose_msg ("init_user_manager");

    return(TRUE);
}
```

```

/*****
*****
*-h- void end_user_manager(max_chans)
* int max_chans; - maximum possible channels
*
* DESCRIPTION
* User defined user_manager cleanup
*
* This termination routine is for multi-threaded user
functions
* where the user to perform certain tasks within the
manager
* process after it has been notified to terminate.
*
* Note that this routine only executes if
number_of_children > 0
*****
*****
*/
void end_user_manager(max_chans)
int max_chans;
{
    verbose_msg ("end_user_manager");

    return;
}

/*****
*****
*-h- void timeout_function(function_code, current_channel,
* number_of_input_buffers,
input_buffer_array,
* p_number_of_output_buffers,

```

```
output_buffer_array)
* DESCRIPTION
*     Function to execute when timeout occurs.
*
* RETURN: nothing

*****
*****
*/
void timeout_function(function_code, current_channel,
                      number_of_input_buffers,
input_buffer_array,
                      p_number_of_output_buffers,
output_buffer_array)
int     function_code;
int     current_channel;
int     number_of_input_buffers;
char    input_buffer_array[MAX_BUFS] [MAX_BUF_SIZE+1];
int     *p_number_of_output_buffers;
char    output_buffer_array[MAX_BUFS] [MAX_BUF_SIZE+1];
{
    printf("Timeout function executed!\n");
}
```

Advanced user function techniques

If you are familiar with writing user functions, you may find the techniques described in the remainder of this chapter useful.

Task Manager

If you find the performance of your user function unacceptable, you can distribute its processing load. The Task Manager feature distributes incoming user function requests to one of several concurrent processes (children), instead of acting on each request sequentially.

This greatly reduces the potential bottleneck associated with a user function that interacts with a slow resource, such as a device or large database. Note that child processes can not exchange data with each other, only with IVR Generator 2.0/S.

The maximum number of children that you can use to defer load is ten. While the system does not limit the number of applications using a user function, only ten channels can use a specific user function simultaneously. Each of the ten child processes handles a single channel request in this situation. All others will wait in a queue.

Note: Degradation can occur when the user function uses more than five concurrent processes.

To designate the number of `user_function` children, edit the variable `number_of_children` at the top of the `usr.c` file. The default is 0, which creates a single-threaded environment (that is, one that spawns no children and processes all requests itself).

You can also include two optional functions in your “C” code:

- `init_user_manager()` performs initialization one time at the creation of the user function task manager (when the first application that uses the user function is loaded).
- `end_user_manager()` performs shutdown tasks in the user function task manager one time (when the last application that uses the user function is unloaded).

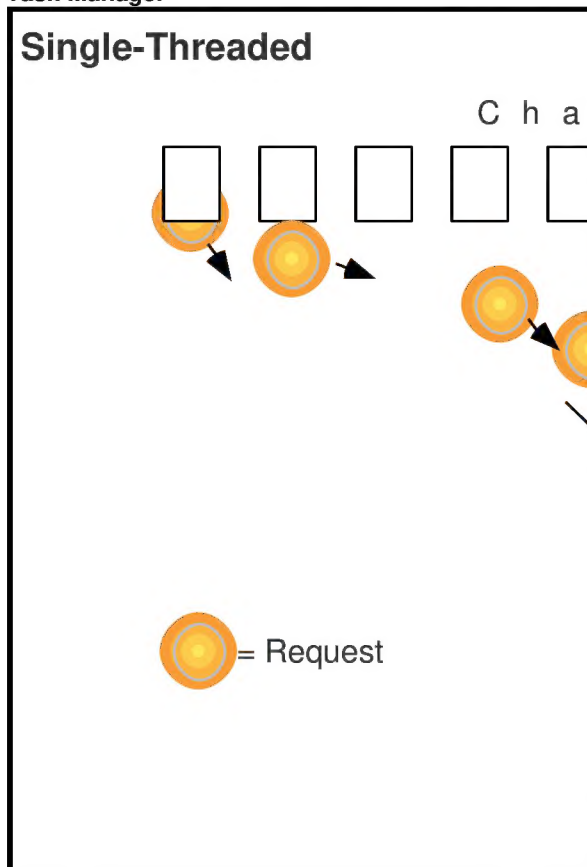
You can use the following two functions in the single-threaded environment, but when used in conjunction with a task manager with concurrent processes, they take on the following specialized purpose:

- `init_user_function()` performs initialization one time at the creation of each user function child process, when the first application is loaded.
- `end_user_function()` performs shutdown tasks in the user function child process one time, when the last application is unloaded.

The dispatching algorithm used by the task manager guarantees that, while a child concurrent process can serve more than one channel, it will consistently serve the same set of channels.

Since a child may need to maintain information about calls associated with each channel, it only needs to dynamically allocate memory for those channels with which it is associated. Therefore, when using multiple child processes (five, for example), each child is responsible only for its part of the memory allocation (in this case, 20 percent)—see [Figure 5-3](#).

Figure 5-3
Task Manager



Time-out

You can set a parameter in your user function to follow a time-out branch if its execution time exceeds the time-out value specified on the parameter page of the USER cell.

To implement this functionality, you must issue a `start_timer (duration)` function in your `user_function` at the beginning of the code you want to bound with a time limit. If the code you are limiting exceeds the duration specified, the system will execute a handler function which will set the global variable `timer_expired` to `TRUE`. The instruction following the code that could exceed the time limit should contain a test for this variable. Be sure to include `stop_timer` in the `user_function` to shut off the timer that was previously requested.

If the `timer_expired` variable is `TRUE`, you can call the function `timeout_function` (which contains code you have specified) to cleanup from a time-out condition before returning a reply to the application.

For example, to open a connection to an external device and cancel it if it exceeds a specified time limit, include the following lines in your `user_function`:

```
start_timer(tout); /* tout is timeout duration */
open_device(...); /* this represents any operation you want
to limit */
if (timer_expired)
/* Check global variable. TRUE means a timeout occurred. */
{
    /* to clean up */

    timeout_function(function_code,
current_channel,
number_of_input_buffers,
input_buffer_array,
p_number_of_output_buffers,
output_buffer_array);
    return  usr_timeout_code();
}
stop_timer(); /* cancel the timer if connection is made within
the time allowed*/
```

Note: The timeout signal will not interrupt `Unix System()` calls or any child processes of the user function.

Use the `timeout_function()` to set up return values to the application. The timeout reply message is sent elsewhere.